A Decentralized Cloud Management Architecture based on Application Autonomous Systems

Dapeng Dong, Huanhuan Xiong, Gabriel G. Castañé, and John P. Morrison

Department of Computer Science, University College Cork
T12 YN60, Cork, Ireland,
{d.dong, h.xiong, g.castane, j.morrison}@cs.ucc.ie

Abstract. Driven by the successful business model, cloud computing is evolving rapidly from a moderate size data center consisting of homogeneous resources to a hyper-scale heterogeneous computing environment. The evolution has made the computing environment ever-increasingly complex, thus, raises challenges for the traditional approaches for managing a cloud environment in an efficient and effective manner. In response, a decentralized system architecture for cloud management is introduced. In this architecture, the management responsibility and resource organization in a conventional cloud environment are re-considered. The reconsideration results in composing a cloud environment into three entities including the Infrastructure, the Cloud Utility and Information Base, and Application Autonomous Systems. In this configuration, service providers focus on providing connected physical resources and introducing featured resources. Information related to the Infrastructure is stored and periodically updated in the Information Base. A consumer employs an Application Autonomous System for managing the life-cycle of a cloud application. An Application Autonomous System in the context of this paper is defined as a self-contained entity that encapsulates a cloud application, the associated resources and the management functions. An Application Autonomous System uses the Information Base and Cloud Utilities to locate and acquire desired resources, subsequently resources are deployed on the Infrastructure by invoking Cloud Utilities. Thereafter, the Application Autonomous System manages the life-cycle of both the application and the associated resources. Consumers are offered opportunities to employ preferred algorithms and strategies for this management. Thus, the responsibility of cloud application management and partially the resource management has shifted from service providers to the consumers in this decentralized system architecture.

Keywords: cloud architecture, decentralized management, resource management, service management

1 Introduction

The success of the business model and the service model of the utility computing have motivated service providers to build and expend their data centers

to an unprecedented size. It has been estimated that Google data centers may consist of one million servers in 2013 [8] and grew to ∼2.5 million servers in 2016 [4]; Facebook data centers consist of ∼60K servers in 2010 [6]; and a more recent Microsoft data center has the capacity to host ∼224K servers on a single site [5]. At the same time, modern data center servers are built with tens of processing cores and hundreds of Gigabytes of system memory, the actual number of virtual machines and/or containers deployed in a data center can be several magnitude more than the the physical servers. Along with the emerging trends for supporting High-Performance Computing (HPC) applications, a wide variety of heterogeneous hardware resources have been introduced to the cloud environments. The management of such large scale and diverse resources becomes increasingly challenging for cloud service providers.

Currently, the majority of existing cloud management platforms can be categorized by the cloud service models, namely Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS), defined by the National Institute of Standards and Technology (NIST) [14]. The cloud platforms for managing IaaS, for example OpenStack [16], provide tools and utility libraries for managing physical and virtual resources. The main challenges for managing IaaS are on the resource allocation efficiency and infrastructure operational efficiency from a technical perspective. The IaaS management platforms also provide Application Programming Interfaces (APIs) and user interfaces to the resource consumers. Consumers use these interfaces for provisioning resources in a self-service mode. For instance, a consumer (either an user or a program) can acquire resources through the interfaces provided by the management platform. The resource scheduling and allocation components of the management platform decide where the resources (e.g., virtual machines or containers) should be allocated. This poses two concerns. First, resources are provisioned and allocated before the deployment of the actual cloud services. This process does not consider the characteristics of each individual service nor the inter-relationships between services that all together constitute as a complete cloud application. This can potentially be harmful to the overall performance of the cloud application, due to that the underlying virtual infrastructure/resources were not constructed/provisioned in an optimal configuration. Second, in response to the quantity and diversity of the underlying resources to be managed, the increasing complexity of resource acquisition requirements, the more restricted service level agreement, the volume of requests for resources and the dynamicity of the environment, novel management strategies for efficient and effective provisioning and managing the life-cycle of resources (physical and virtual) are needed, which determines a sustainable cloud environment.

The subscribers of IaaS services are responsible for configuring the leased resources and the subsequent deployment of the services/applications on the resources. Often, the configuration processes and the deployment of services/applications are time consuming and may require domain-specific knowledge and skills. To ease these processes, management frameworks and platforms for PaaS start gaining popularities, for example, OpenStack Solum [19] and Apache Brook-

lyn [1]. These PaaS management platforms provide facilities for consumers to express their needs and requirements in a blueprint alike style, articulated in domain-specific languages, for example, Topology and Orchestration Specification for Cloud Applications (TOSCA) [20] and Cloud Application Management for Platforms (CAMP) [3]. These languages are sufficiently flexible to express the details of the entire service and resource life-cycle management, and surely results in a blueprint that is complex and subject to error-prone.

Nevertheless, the service and resource deployment are still two separate processes. Resources are provisioned and deployed by invoking IaaS management functions, for example, in an OpenStack managed environment, an application and resource orchestration framework, such as Heat [9], invokes OpenStack Nova services (e.g., nova-api, nova-conductor, nova-scheduler and nova-compute) for provisioning and deployment of virtual machines [15], and uses OpenStack Neutron for creating virtual networking environment [17]. Given that the underlying resources are ready to use, the Heat deploys services/applications on the resources. This also implicitly allows cloud consumers to have full control over the management of applications as well as the underlying resources and subsequently narrows down the opportunities for cloud service providers to improve resource utilization, power efficiency and potentially the quality of services. Note that SaaS is often built on top of PaaS and IaaS. SaaS has the main focus on providing functions, utilities and services to consumers, directly. Thus, SaaS is outside of the context of this paper.

Additionally, the IaaS/PaaS resource allocation components typically do not take the characteristics of the services into account when provisioning resources. The optimizations are generally carried out afterwards during the service/application lifetime. Such optimizations are traditionally done through monitoring various aspects of resource usages, such as processor utilization, memory utilization, and network bandwidth consumption. But, this is often done for the interests of service provides, for instance, virtual machine consolidations for improving server utilizations. Certainly, more restrictions and requirements (for both consumers and service providers) can be expressed in a blueprint, provided that the service descriptionlanguages are capable of doing so. As the size of the data center increases and the number of services/applications hosted by the data center grows rapidly, the management overhead associated with such optimization becomes non-negligible. Shifting such overheads to cloud consumers may results in a more sustainable environment. In other words, shifting the management responsibility to consumers and splitting the centralized management overheads into distributed management on a per application basis. In the context of this paper, a cloud environment is virtually divided into Application Autonomous Systems (AASs). Each AAS presents a self-contained management domain and logically manages a cloud application. In this configuration, consumers need to bear the cost for the management processes (the underlying resources that are needed to host the management functions) and cloud service provides only need to provide resources and a set of common utilities that are essential for an AAS to function.

An AAS interprets and executes an application Blueprints consisting of many services and taking into account of the entire collection of services to determine an optimal set of resources, and subsequently controls the application and resource life-cycle management. It is also possible for an AAS to be reused for the similar type of applications. In this respect, it is imperative to maintain a separation between application life-cycle management and resource management. Thus, an AAS can address the potential conflicts between cloud service management and cloud resource management while maximizing user experience and cloud efficiency on each side, as well as making it is possible to implement continuous improvement on resource utilization and service delivery.

The remainder of this paper is organized as follows. Backgrounds and discussions on several related works are given in Section 2. The proposed solution is introduced in Section 3, Important concepts and detailed architecture are given in Sections 4. Future directions and conclusions are drawn in Section 5.

2 Background and Related Work

Existing IaaS/PaaS management platforms manage the life-cycle of cloud applications together with their associated underlying resources. Three representative platforms are used in this section to highlight the mainstream approaches for managing a PaaS/IaaS cloud environment. Fig. 1 shows the application/resource life-cycle management schemes employed by the OpenStack Solum [19], Apache Brooklyn [1], and OpenStack Heat [9]. These platforms provide tools for deploying and managing services/resources, and provide APIs to interface with cloud consumers and/or applications. Solum and Brooklyn are usually considered to be PaaS management platforms, while Heat is an service/resource orchestration framework for IaaS.

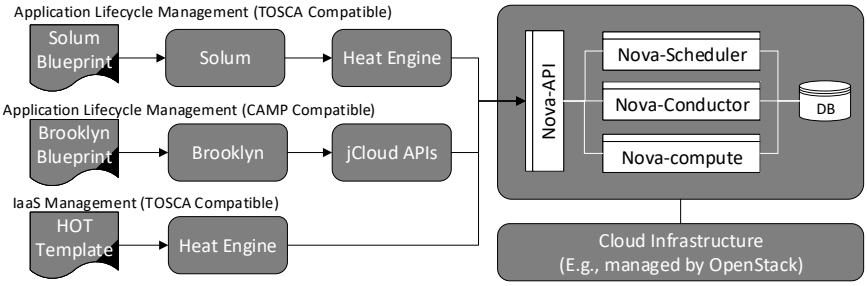

Fig. 1. An overview of cloud application/resource life-cycle management in OpenStack Solum, Apache Brooklyn and OpenStack Heat.

The Solum and Brooklyn frameworks allow cloud consumers to deploy and execute blueprints written in a service description language, particularly, TOSCA

and CAMP. These languages are used to describe the characteristics of application components, deployment scripts, dependencies, locations, logging, policies, and so on. The Solum engine takes a blueprint as an input and converts it to a Heat Orchestration Template (HOT), this template can be understood by the application and resource management engine (Heat). The Heat engine, thereafter, carries out the application and resource deployment by invoking the corresponding service APIs that are provided by the underlying cloud infrastructure framework, for example, Nova and Neutron APIs.

In contrast, Brooklyn engine converts a blueprint into a series of jCloud [11] API calls that can be used to interact with the underlying cloud infrastructure management components. For example, a jCloud API call for creating an virtual machine in OpenStack is sent to the *nova-api* component. The *nova-api* component notifies the *nova-scheduler* component to determine where the requested virtual machine should be created. Once an suitable server is identified, the request is forwarded to the *nova-compute* component to carry our the actual deployment on the selected server. This "*Request and Response*" approach is simple, robust, and efficient. However, it should be noted that each request is processed independently, making it impossible to consider relative placement of virtual machines associated with multiple requests. Additionally, this "*Request and Response*" approach does not support the optimal deployment for a group of services that all together are considered as a complete cloud application. This limitation is not specific to virtual machine placement, but also applies to the deployment of containers, for example in a Kubernets [2] [18] or Mesos [10] managed containerized environments.

3 Architecture Overview

Conventional clouds provide interfaces to consumers for consuming resources in a self-service manner. Either in an IaaS or a PaaS model, beneath the user interfaces, the underlying resource management typically take a centralized management approach. Recall from the discussions given in Section 1 and 2 that due to the ever-increasing size of the data center and resource heterogeneity, the centralized resource management systems are continuously being challenged. In response, a decentralized management architecture is introduced, as shown in Fig. 3. The main design principle is to divide a cloud environment into three entities including the Infrastructure, Cloud Utilities and Information Base, and Application Autonomous Systems. The Infrastructure provides interconnected physical resources. Information related to resources, such as server status and computational resource availabilities, are stored and periodically updated in the Information Base. An AAS is a self-contained entity that encapsulates a cloud application, the associated resources and the management functions. AASs use the Information Base and Cloud Utilities to locate and acquire resources, and resources are deployed on the Infrastructure by invoking the Cloud Utilities. Thereafter, the AAS manages the life-cycle of both the application and the associated resources.

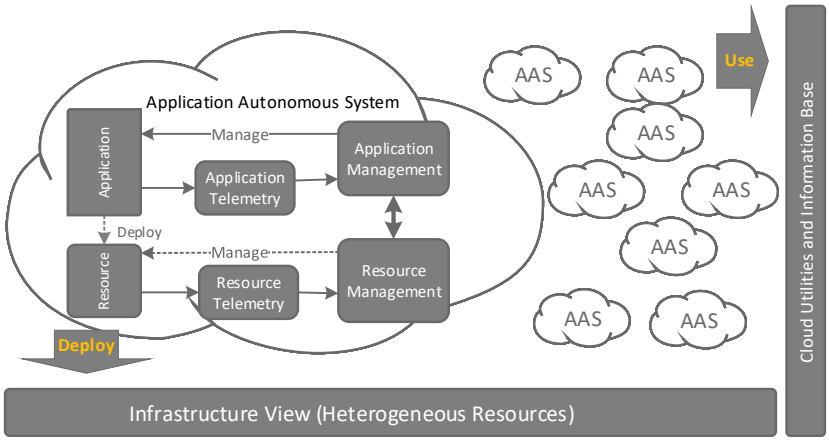

Fig. 2. Decentralized system architecture based on Application Autonomous Systems.

In this design, the centralized resource management is divided by the number of AASs. Each AAS makes its own decisions on what resources to be used and where to provision the resources. This gives an opportunity to the consumers to employ preferred strategies for the management of their applications and resources. In addition, since each AAS manages a relatively small number of services, more sophisticated management strategies and optimization methods can be employed. All AASs indirectly compete with each others for the best of resources. This implicitly shifted the management costs and responsibilities from service providers to consumers. As all information about the resources are logged into the Information Base, an AAS can query the Information Base with desired features to locate appropriate resources. This also allows service providers to focus on providing better quality resources and makes the Infrastructure more *static*. When the number of AAS increases, it only makes AASs harder to compete with each others for resources. This has no effects on the underlying resources, the Cloud Utilities, and the Information Base that are organized by the service provider. At the same time, when adding more resources and/or introducing different types of resources to the Infrastructure, AASs are not affected. New features, such as computation accelerators, are advertised to AASs. It is the AASs' responsibility to locate the featured resources. Thus, a cloud environment that employs the decentralized management becomes more sustainable.

Since a cloud environment is logically divided into a number of Application Autonomous Systems, the answers to what defines a management domain for an AAS, how a management domain can be constructed, and how an AAS evolves internally and externally with the environment to achieve the designated

goals, ensures the proposed decentralized management approach to function in an efficient and effective manner.

4 Application Autonomous System

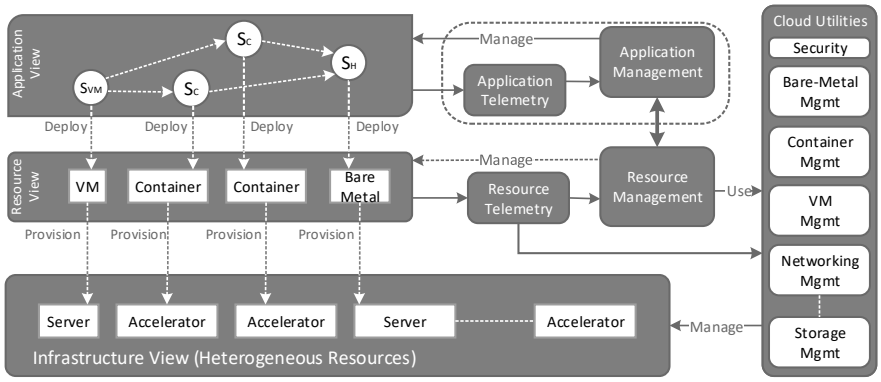

Fig. 3. The internal structures of an Application Autonomous System and its relationships to the cloud environment.

An Application Autonomous System is an independent entity. An AAS manages a group of services that can be logically grouped together to form a complete cloud application. An AAS also manages the resources that are associated with the managed cloud application. AASs do not have direct intercommunications with each others. Each AAS reacts upon the changes in the cloud application and the environment. Conceptually, the environment is the Infrastructure. The changes in the environment is the changes of the status of the infrastructure, for example, the changes of the status on the computational resource availability of each server and/or the average networking traffic load on a particular link. The Resource Management component interacts with both the Application Management and the Infrastructure. Thus, the Resource Management must employ algorithms/strategies that can satisfy both the consumers' and service providers' interests. In contrast, the Application Management is an optional component for managing applications at various levels. In the absence of the application-specific interfaces, the Application Management manages the life-cycle of the application (e.g., deploy and decommission). With provided application-specific interfaces, more advanced optimizations can be carried out (e.g., load-balancing).

The AAS-based management approach provides PaaS services. It must be noted that the definition of the platform is a broad term. It can be a management framework, such as Apache Brooklyn, an application server, such as Google

App Engine, or an analytic platform, such as Hadoop/Spark. The AAS-based management approach can be considered as an management framework, such as Apache Brooklyn. Application servers or analytic platforms can be seen as cloud applications in this context. However, the differences between AAS and the Brooklyn alike frameworks lie on the cloud application and resource management styles. More specifically, the cloud application and resource management in Brooklyn alike frameworks are tightly coupled. In other words, cloud applications and the associated resources coexist. Decommission of a cloud applications implies freeing the underlying resources. In contrast, AAS is designed based on the concept of Separation of Concerns [7], i.e., cloud applications and the associated resources are manged independently, but they are also complementary to each other.

4.1 The Concept of Separation of Concerns

The main idea of the Separation of Concerns is to decouple the cloud application management from the associated resource management, while the *desires* (e.g., needing for more resources) from cloud application management actions can be forwarded asynchronously to the resource management functions, meaning that the resource management functions can decide whether to react upon receiving a *desire* based on the feasibility of doing so. Inversely, the outcomes from a resource management action (e.g., virtual machine migration to avoid resource contentions or server consolidation for improving power efficiency) can be fed back to the cloud application management functions in the same asynchronous manner. The separation yields several unique features. First, resources do not have to rely on the existences of cloud applications. When a cloud applications is at the end of its lifetime, the underlying resources can be kept by the AAS, so that the AAS can be reused as a pre-provisioned template for incoming cloud applications that have similar characteristics and requirements on resources, thus, it can accelerate the service delivery processes and improves user experiences. Second, the separation allows the cloud application and resource management functions to focus on their respective optimizations. Third, cloud application optimization and management generally require application-specific interfaces to interact with. More often, these interfaces are not available for many existing cloud deployable applications. In such a case, the absence of the application-specific interfaces does not affect the deployment and execution of the cloud application.

4.2 Resource Management in Application Autonomous Systems

A striking characteristic of traditional cloud management platforms is apparent, that global optimizations between multiple services are not generally available due to the way in which resource requests are individually processed. The Separation of Concerns provides direct architectural support for considering optimal resource requests from multiple interacting services simultaneously.

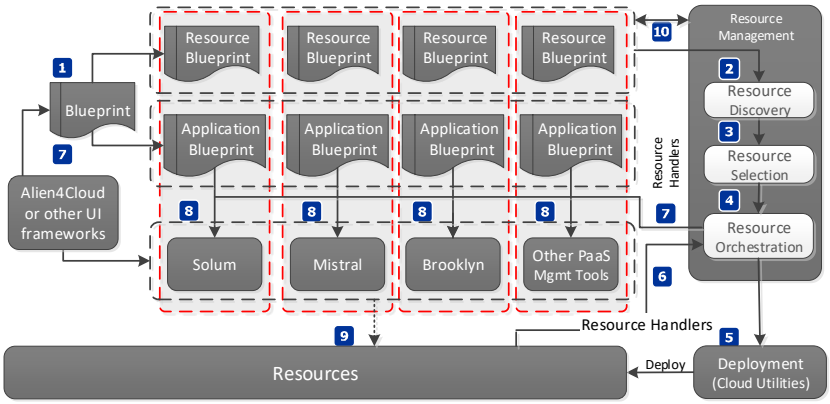

Fig. 4. Resource management in an Application Autonomous System.

In order to separate the concerns of cloud application life-cycle management and resource life-cycle management, a cloud application, especially when a cloud application consists of several dependent services, need to be expressed in a service description language, for example, TOSCA and CAMP, in a *blueprint* style. In the context of AAS, a cloud application blueprint deployment starts by decomposing a blueprint into two parts: Resource Blueprint and Application Blueprint which can be used by the AAS resource/application management, respectively, as shown in Fig. 4 (label 1). The Resource Blueprint is first sent to the Resource Discovery and Resource Selection components to locate and acquire the most appropriate (defined by constrains, parameters, and preferences of both users and systems) resources for the cloud application, as indicated in Fig. 4 (label 2 & 3). The returned location information and the Resource Blueprint are then sent to the Resource Orchestration engine (e.g., a customized Heat Engine) to carry out the actual resource deployment on the infrastructure. The deployment processes are essentially invoking the Cloud Utilities, for example, in a Kernel-based Virtual Machine (KVM) [12] managed virtual environment, provisioning virtual machines on a designated server requires to invoke a series of *libvirt* API calls. These *libvirt* APIs thus must be included as a part of the Cloud Utilities. The resource deployment process results in the return of a number of resource handlers (A resource handler can be a login account includes, for example, user name, access key, and IP address to a virtual machine). These resource handlers are sent back to the Resource Orchestration engine, which, in turn, will use them to finalize the Application Blueprint. The Application Blueprint is then forwarded to the corresponding application life-cycle management component to carry out the application deployment on the pre-provisioned resources. This process is shown in Fig. 4 (label 6, 7, 8 and 9).

In contrast to existing frameworks, the proposed service delivery model will facilitate blueprint developers to specify comprehensive constraints and quality of service parameters for both services and resources. Based on the specified constraints and parameters, in contrast to existing solutions, can provide an initial optimal deployment of the resources. For example, creating and identifying resources on adjacent physical servers to minimize communication delay or provisioning containers with attached GPUs to balance performance and cost. During the life-cycle of the resources, optimizations (e.g., load-balance and elasticity) are done by the Resource Management functions, as shown in Fig. 4 (label 2, 3 and 4), in-conjunction with the resource telemetry services, as shown in Fig. 3, in a closed feedback-react loop.

4.3 Cloud Application Management in Application Autonomous Systems

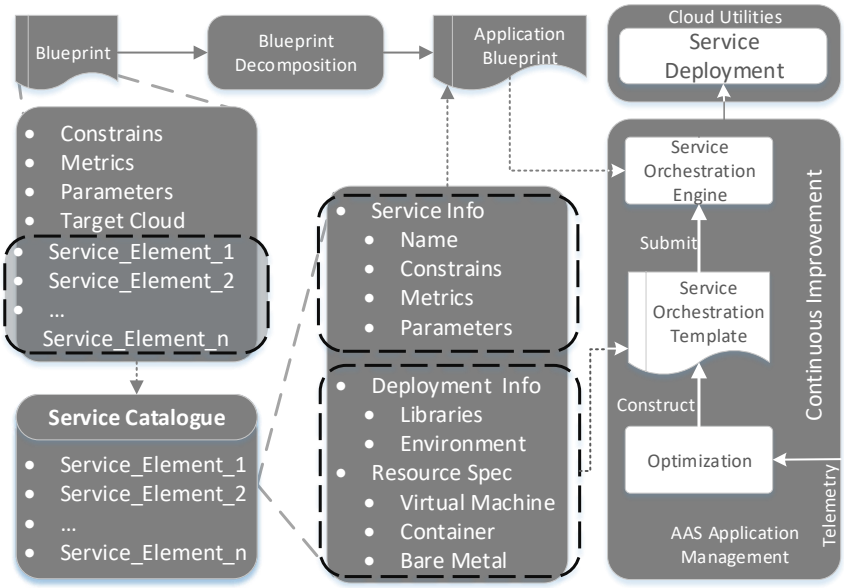

Fig. 5. Application management in an Application Autonomous System.

The cloud application deployment is an incremental process. Depending on the different types of resources and the resource availabilities, each individual resource provisioning process may take different time to complete. For instance, given a blueprint that requires a virtual machine and a container resources,

provisioning a virtual machine may take several tens of seconds, where as provisioning a container may only take several seconds. In order to improve the service delivery experiences, the resource handlers are returned asynchronously. Upon receiving a resource hander or a group of resource handlers, a temporary Application Blueprint is constructed, as shown in Fig. 4 (label 7 & 8). The temporary Application Blueprint is then sent to the Service Orchestration engine for deployment, as shown in Fig. 5. Subsequently, the Service Orchestration engine invokes the Cloud Utilities for the actual cloud application deployment. It must be noted that all deployment related information is embedded in the Application Blueprint, as shown in Fig. 5.

The Optimization component together with the Application Telemetry services (as shown in Fig. 3) attempt to perform continuous improvement over the life-time of the deployed blueprint. This is achieved by periodically reconstructing an Application Blueprint based on the information received from the Application Telemetry service, and re-submit the updated Application Blueprint to the Service Orchestration engine for the execution of the optimization actions, such as, load-balancing.

The concept of the Separation of Concerns has been realized in the CloudLightning project [13]. In the service provider-consumer context, CloudLightning defines three actors including End-users (application/service consumers), Enterprise Application Operator/Enterprise Application Developer, and Resource Provider. These actors represent three distinct domains of concerns.

- For the end-users, the concerns are cloud application continuity, availability, performance, security, and business logic correctness.
- For the Enterprise Application Operators/Enterprise Application Developers, the concerns are cloud application configuration management, performance, load balancing, security, availability, and deployment environment.
- For the Resource Providers, the concerns are resource availability, operation costs such as power consumption, resource provisioning, resource organization and partitioning.

CloudLightning is built on the premise that there are significant advantages in separating these domains and the use of service description languages has been designed to facilitate this separation. Inevitability, there will always be concerns that overlap the interests of two or more actors. This may require a number of actors to act together, for example, an Enterprise Application Operator may need to configure a load-balancer and a Resource Provider may need to implement a complementary host-affinity policy to realize high-availability. These overlapping concerns are managed by each individual Application Autonomous Systems by providing vertical communications between the application life-cycle management and the resource life-cycle management.

Enterprise Application Operators/Enterprise Application Developers are responsible for managing the life-cycle of Application Blueprints. At the same time, the underlying resources are managed independently by the Resource Provider. As a result, the following advantages accrue:

- continuous improvement on the quality of the Blueprint services delivery;
- reducing the time to start a service and hence improve the user experience by reusing resources that have already been provisioned;
- resource optimizations and energy optimization;
- creating a flexible and extensible integration with other management frameworks such as the OpenStack Solum or Apache Brooklyn management system.

The first step in the CloudLightning is to establish a clear services interface between the service consumer and the service provider. The essence of this interface is the establishment of a separation of concerns between cloud service consumers and cloud service providers. In this view, various service implementation options can be assumed to already exist and consumers no longer have to be an expert creator of those service implementations. Consumers should not have to be aware of the actual physical resources being used to deliver their desired service, however, given the fact that multiple diverse implementations may exist for each service (each on a different hardware type, and each characterized by different price/performance attributes) consumers should be able to distinguish and choose between these options based on service delivery attributes alone. Service creation, in the approach proposed here, remains a highly specialized task that is undertaken by an expert.

5 Conclusion

The Application Autonomous System based on the decentralized cloud management tries to re-align the evolving cloud environment with the services-oriented architecture of conventional clouds. Application Autonomous Systems management uses a vertical management approach that implements the concept of Separation of Concerns. It is a more sophisticated management approach than current self-service models. The implementation allows the application management and the resource management to operate independently, consequently, it separates consumer concerns with optimizing cloud applications and service provider concerns with the efficient use of resources and the reduction of operational costs. Application Autonomous Systems virtually and logically divide a cloud environment in to a number of self-contained management domains, hence, it represents a decentralized system architecture. The application and resource decentralization shift the management responsibility from cloud service providers to consumers. This makes a cloud service provider focusing on providing resources, and consumers on taking responsibility for managing applications, thus, it results in a more sustainable computing environment.

Acknowledgment

This work is funded by the European Unions Horizon 2020 Research and Innovation Programme through the CloudLightning project under Grant Agreement Number 643946.

References

1. Apache Brooklyn: https://brooklyn.apache.org/ (2017)
2. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016)
3. Carlson, M., Chapman, M., Heneveld, A., Hinkelman, S., Johnston-Watt, D., Karmarkar, A., Kunze, T., Malhotra, A., Mischkinsky, J., Otto, A., et al.: Cloud application management for platforms. OASIS, http://cloudspecs. org/camp/CAMP-v1. 0. pdf, Tech. Rep (2012)
4. Data Center Knowledge: Google Data Center FAQ. http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq/ (2017)
5. Data Center Knowledge: Special Report: The Worlds Largest Data Centers. http://www.datacenterknowledge.com/special-report-the-worlds-largest-data-centers/ (2017)
6. Data Center Knowledge: The Facebook Data Center FAQ. http://www.datacenterknowledge.com/the-facebook-data-center-faq/ (2017)
7. Dong, D., Xiong, H., Morrison, J.: Separation of concerns in heterogeneous cloud environments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER, pp. 775–780 (2017)
8. Ghiasi, A., Baca, R., Quantum, G., Commscope, L.: Overview of largest data centers. In: Proc. 802.3 bs Task Force Interim meeting (2014)
9. Heat: https://github.com/openstack/heat (2017)
10. Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A.D., Katz, R., Shenker, S., Stoica, I.: Mesos: A platform for fine-grained resource sharing in the data center. In: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11, pp. 295–308. USENIX Association, Berkeley, CA, USA (2011)
11. jCloud: https://jclouds.apache.org (2017)
12. Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A.: kvm: the linux virtual machine monitor. In: Proceedings of the Linux symposium, vol. 1, pp. 225–230 (2007)
13. Lynn, T., Xiong, H., Dong, D., Momani, B., Gravvanis, G.A., Filelis-Papadopoulos, C.K., Elster, A.C., Khan, M.M.Z.M., Tzovaras, D., Giannoutakis, K.M., et al.: Cloudlightning: A framework for a self-organising and self-managing heterogeneous cloud. In: CLOSER (1), pp. 333–338 (2016)
14. Mell, P., Grance, T., et al.: The nist definition of cloud computing (2011)
15. Nova, O.: http://docs.openstack.org/developer/nova/ (2017)
16. OpenStack: The openstack project. https://www.openstack.org (2011)
17. OpenStack Neutron: https://github.com/openstack/neutron (2017)
18. Rensin, D.K.: Kubernetes - Scheduling the Future at Cloud Scale. 1005 Gravenstein Highway North Sebastopol, CA 95472 (2015)
19. Solum: https://github.com/openstack/solum (2017)
20. TOSCA, O.: Topology and orchestration specification for cloud applications (tosca) primer version 1.0 (2013)